Nicholas Chan
nipchan@ucsc.edu
4/25/2021
CSE13s Spring '21

Design Document:
Assignment 3: Sorting

Sorting algorithms are often applied to arrays or array-like structures such as stack/queue abstract data types. Stacks and queues are structures that include arrays along with additional data that cannot be held by an array alone.

In assignment 3, I have been given the task of implementing bubble sort, shell sort, quicksort with a stack and quicksort with a queue along with a testing harness to demonstrate each sorting algorithm. Statistics from each sort such as the counts of moves, swaps and comparisons will be made available in the output of the testing harness. The testing harness will take in the command line argument [-absqQ] [-n length] [-p elements] [-r seed]. The first argument option corresponds to all sorting algorithms, Bubble sort, Shell sort, quicksort with a stack and quicksort with a queue. -n, -p and -r are explained by the value type they take in, with n being array length to use, p being the number of elements to display from the sorted array and r being the random seed to set.

Putting program together/Makefile functionality:

- .c files depend on .h files
- Must generate .o files from all .c files associated with sorts, stacks and queues
    - These .o files will be linked to the test harness .
- bubble.h
    - Will require bubble.c for use in test harness
- shell.h
    - Will require shell.c for use in test harness
    - Will include gaps.h
        - Actually included in shell.c
    - shell.c will be compiled into shell.o which will be linked in the compilation of sorting.o into an executable file.
- quick.h
    - Will require quick.c for use in test harness
    - Will include stack.h and queue.h
        - These are actually included in quick.c
    - quick.c will be compiled into quick.o which will be linked (along with stack.o and queue.o) in the compilation of sorting.o into an executable file.

Pre Lab Questions:

Part 1: Bubble Sort

1. It would take 6 passes through the array (from left to right) with 10 total swaps made over 21 comparisons to sort the array into ascending order.
2. I would expect 21 comparisons are possible in the worst case scenario. This is true using the equation n(n+1)/2 when doing a bubble sort to get an array in ascending order into descending order.

Part 2: Shell Sort

1. If the size of the gap for shell sort was 1 and it was the only gap in a gap sequence, shell sort would function at its worst time complexity. This is because smaller gap sizes cause shell sort to perform more comparisons. This makes sense as using a gap size of 1 in a shell sort would make it identical bubble sort. The time complexity of a shell sort would be even worse when the array to be sorted is in the opposite order that you are trying to sort it into (like having an ascending order array when you are trying to sort it in descending order) in addition to having a gap size of 1. Shell sort's time complexity can be improved by choosing an optimal sequence (sequence will end with 1) of gaps to cycle through in a sort. This sequence is dependent on the size of the array to be sorted, but formulas to determine good sorting sequences to use exist (I found a list of formulas for sequences on wikipedia)

Part 3: Quicksort

1. Quicksort isn't doomed because its time complexity is reliant on the pivot used, which means that the pivot to where you form partitions can be changed to avoid the worst case time complexity situation. The best situation is if your partitions form (almost) equal halves for a quick sort. The worst case scenario would be when a partition with only one element in it and another partition with the rest of the elements are chosen for a quick sort (having the most possibly imbalanced partitions for a quick sort). A way to avoid the worst case scenario for quicksort would just be to pick a pivot that isn't any of the extreme values of an array to be sorted. Eugene's section helped make these characteristics of quick sort clear to me. The wikipedia page on quick sort also helped me understand how quick sort is a type of divide and conquer algorithm and how that characteristic was tied in with partitioning and sorting performance.

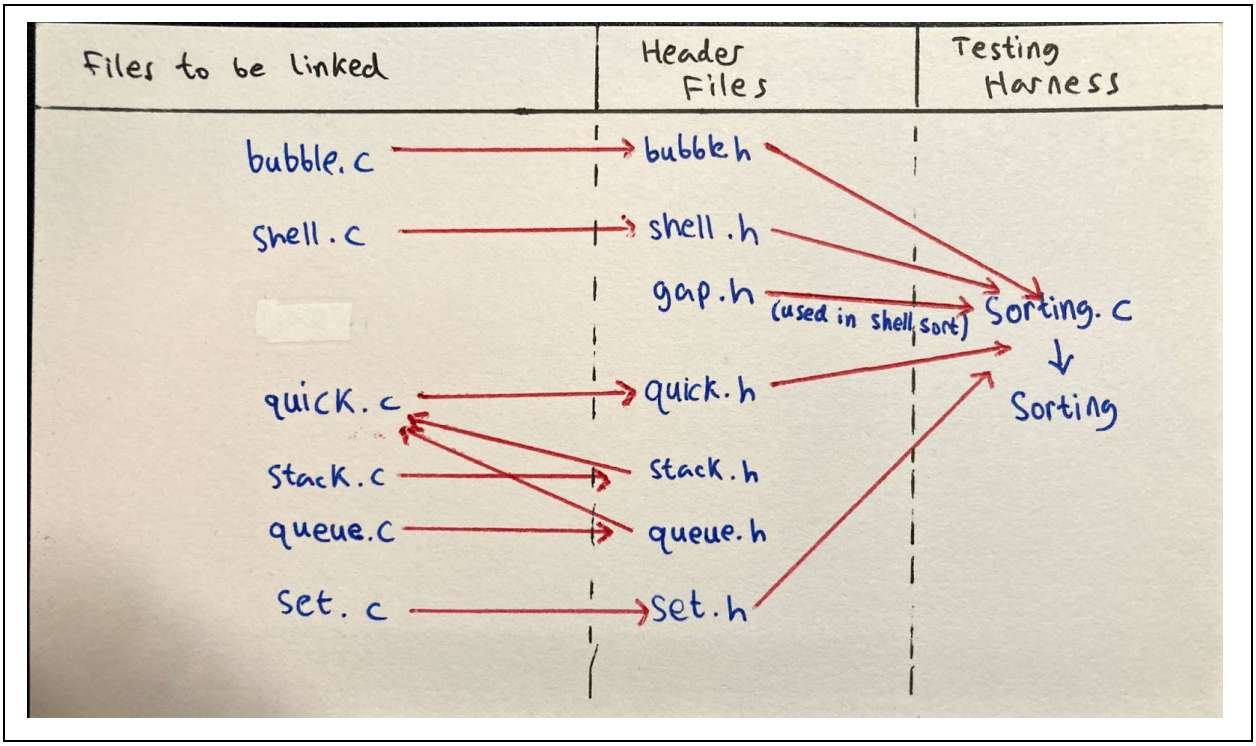
Part 4:

I plan on adding a new header file which would contain extern variables to hold the counts for moves and comparisons for each of my sorting algorithms.

- I will only count comparisons whenever two array items are used to determine a swap in a sorting algorithm. This situation would usually occur when a conditional statement occurs in the main sorting loop of a sorting algorithm.
- I will keep track of the number of moves made by incrementing a counter for the number of moves for a sorting algorithm only when array values are:
    a. Assigned to a temporary position
    b. Reassigned to an array pointer (either previously from a temporary position or position on the array)

Top Level

How my Files will Interact



Pseudocode

Note:
Pseudocode is heavily inspired from the asgn3 instruction doc python pseudocode. I include curly braces to help me keep track of scopes.

Bubble Sort

```
bubble_sort(an array of integers)

Store integer for length of the array
Store a bool for status as swapped

While status for swapped is true
   Set status for swapped to false (will be changed if there is a need for swapping)
   For each item past the first in the input array
      If item[n] < item[n-1]
         Swap item[n] with item [n-1]
         Set swapped status back to true
   Decrement value storing length of array as last item has been correctly positioned
```

Shell Sort

```
shell_sort(array on integers, gap sequence)
   For each gap in the given sequence of gaps (will be using Pratt Sequence)
      For each index in the span of a given gap's value to the length of the input array -1
         Store a given index
         Store array[index] (the array item of the index)
         While the given index >= given gap and array[index] < array[index-gap]
            Array[index] = array[index-gap]
            Array[index-gap] = array[index]
            Index is now made into the index minus the gap
            Array[index] now takes the value of the stored array[index] item
```

Quick Sort

Part 1: Partitioning (will be essential for quicksort)

```
partition (array of integers, low value, high value)
   Choose a pivot position (approximately the midpoint of the array)
   Choose a lower bound
   Choose an upper bound
   While lower bound < upper bound
      Increment lower bound by one
      While array item at lower bound is < chosen pivot
         Increment lower bound by one
      Decrement upper bound by one
      While array item at upper bound is > pivot
         Decrement upper bound by one
      If lower bound is < upper bound
          Swap array item of lower bound with array item of upper bound
   Return lower bound value
```

Part 2: Quicksort with either stack or queue

```
quick_sort_stack(array of integers)
   Set a lower bound to 0
   Set upper bound to the last index of the input array
   Generate_stack(size of array)
   Push the lower bound and upper bound to the stack
   While the length of the stack isn't 0
      upper bound is set to the value popped from the stack on a call to pop
      lower bound is set to the value popped from the stack on a call to pop
      partition(input array, lower bound, upper bound)
      If lower bound is < partition
         Push lower bound to stack
         Push partition to stack
      If upper bound is > partition + 1
         Push upper bound to stack
         Push partition + 1 to stack

quick_sort_queue(array of integers)
   Set a lower bound to 0
   Set upper bound to the last index of the input array
   Generate_queue(size of array)
   Enqueue the lower bound and upper bound to the stack
   While the length of the queue isn't 0
      upper bound is set to the value dequeued from the stack on a call to dequeue
      lower bound is set to the value dequeued from the stack on a call to dequeue
      partition(input array, lower bound, upper bound)
```

If lower bound is < partition
    Enqueue lower bound to stack
    Enqueue partition to stack
If upper bound is > partition + 1
    Enqueue upper bound to stack
    Enqueue partition + 1 to stack

Stacks and Queues

Stacks
Contain
    Integer indicating top position
    Integer indicating capacity
    Pointer to an array of integers

Queues
Contain
    Integer indicating head position
    Integer indicating tail position
    Integer indicating size
    Integer indicating capacity
    Pointer to an array of integers

Call to generate stack/queue
Allocate memory for stack/queue structure
    Allocate memory for array of items that will be pointed to by the pointer in each stack/queue

Push/enqueue (stack/queue pointer, integer to insert into stack/queue item array)
If stack/queue does not have enough memory
    Reallocate original stack/queue structure pointer to a block with twice as much memory
Add item into stack/queue item array

Pop/dequeue (stack/queue pointer, new variable to assign popped/dequeued item)
Move top/head pointer from item in array
    Set new variable equal to popped/dequeued item

Destructor(pointer to the stack/queue pointer)
    Free items array pointer in the stack/queue structure and set that pointer to null
    Free pointer to stack/queue structure and set that pointer to null

Design Process

Before beginning this lab, I familiarized myself with bubble sort, shell sort and quick sort. I did this by watching some videos and doing some drawings to visualize the movements of the sorts and then moved onto reading the descriptions of each sort provided in the asgn3 instructions pdf. After this I moved onto walking through the python pseudocode implementations of these sorts and then worked on setting up stacks and queues in c. I believe that all this preparation was necessary to be able to know how to implement these sorts in c.

1. Beginning the programming process, I created a swap function which I would reference in each of my sorting functions.
    a. This swap function was stored in a helper file that I named stast.c and stats.h.
    b. I also stored my statistics collected from each sort job in these helper files
    c. Referencing the swap function in this file was very helpful
2. I then worked on the bubble.c and shell.c
    a. Since both of these programs only required the handling of an array and no extra ADT's, implementing them was fairly straightforward
    b. Walking through the provided python pseudocode helped
3. I then worked on setting up my stack and queue ADTs for quick.c
    a. In stack.c I drew some figures of a stack going through pushing and popping to keep track of the movements of the top pointer, size and array items.
        i. After finishing up a draft of stack.c, I ran the print_stack function to test if pushing, popping and the rest of accompanying functions for stack.c worked
    b. In queue.c I went through a similar process as I did with stack.c
        i. queue.c was pretty similar to stack.c with the exceptions of dequeue, enqueue and how items were circulated in the items array of the queue ADT
        ii. Correctly implementing head and tail were a little tricky because I was a little new to the use of modulo for never exceeding the bounds of the queue's array item pointers
4. I then worked on quick.c, which was virtually the same as the python pseudocode with the exception of my own implementations for the stack and queue ADTs
    a. quick.c was made up of the stack implementation of quicksort and the queue implementation of quicksort
5. With all my functions completed, I worked on the test harness called sorting.c
    a. Eugene's section was really helpful for learning how to correctly use the set.c and set.h files to parse command line arguments.
    b. The test harness was made of my main function which largely consisted of a switch statement and set to keep track of command line arguments and a series of

conditional statements to determine which sort function to call and which command line arguments to pass through.

6. Finally I created a Makefile to properly compile the source code of my test harness and source files.
    a. This called for -lm to link my files for compilation

I thought this assignment and the experience I had with these sorting algorithms was really interesting, although not intuitive at first (from my own experience). I felt that an example in section or class for how to use extern variables would have been pretty helpful for this assignment which required me to store statistics.

Significant References:

- https://en.wikipedia.org/wiki/Quicksort#Algorithm
- https://en.wikipedia.org/wiki/Shellsort#Gap_sequences